



Comprehensive Study of Persistence Techniques in In-memory Databases

Aryan Chaudhari^{1*}, Harsh Bhat², Abhishek Belgaonkar³, Aditya Supare⁴ and Seema Patil⁵

^{1,2,3,4,5}Department of Computer Engineering and Technology,
Dr. Vishwanath Karad MIT World Peace University, Pune, India

¹aryanchaudhari44@gmail.com, ²harsh121102@gmail.com,

³abhishekbelgaonkar23@gmail.com, ⁴adityasupare2003@gmail.com,

⁵seema.patil@mitwpu.edu.in

Abstract. In recent years, NoSQL databases have become essential for delivering Big data web services. As memory capacities increase, there is a heightened focus on In-memory NoSQL (IM-NoSQL) systems, which use dynamic random-access memory (DRAM) to enable minimal latency. However, because DRAM is volatile, IM-NoSQL systems need effective persistence and recovery methods to prevent data loss during server failures. This paper presents a detailed study of the performance of persistence and recovery techniques in IM-NoSQL databases. The evaluation examines the performance of Snapshotting and logging techniques, focusing on their effectiveness in failure recovery. Our research aims to answer critical questions: (i) This study investigates whether IM-NoSQL systems maintain efficiency under memory constraints.(ii) What are the performance trade-offs between using snapshots and logging? (iii) How quickly can an IM-NoSQL system recover after a failure? (iv)How does the choice of persistence method affect the system's performance? This study utilizes Redis as a representative IM-NoSQL system to evaluate persistence strategies, recovery durations, and system performance metrics.

Keywords: In-memory databases, IM-NoSQL, persistence techniques, Redis, NoSQL, Snapshotting, logging, Append-Only File (AOF), key-value stores, RAMCloud, Memcached, Apache Ignite, hybrid persistence models, crash recovery, database benchmarking, cloud-native databases, workload optimization.

1 Introduction

NoSQL - stands for Not Only SQL- databases have become very popular and are widely used for web services managing large datasets. The most popular examples of such systems include Google's Bigtable [3], Amazon's Dynamo [4], the HBase used by Facebook and Yahoo!, and Voldemort implemented at LinkedIn. Recently, developments in memory capacity together with low costs of DRAM unit have enabled the emergence of In-memory NoSQL systems (IM-NoSQL) as an integral part of quite a number of Internet-scale web services that bring not only spectacular

© The Author(s) 2025

S. Bhalerao et al. (eds.), *Proceedings of the International Conference on Recent Advancement and Modernization in Sustainable Intelligent Technologies & Applications (RAMSITA-2025)*, Advances in Intelligent Systems Research 192,

https://doi.org/10.2991/978-94-6463-716-8_83

low latency. On the industrial front, examples such as Memcached, Redis, and partially in-memory database MongoDB dominate, while academic systems like RAMCloud [7], MemC3, FaRM [11], and MICA [14] show advancements in research. The increasing popularity of NVM notwithstanding most IM-NoSQL systems rely still on volatile DRAM.

The use of DRAM makes persistence and recovery mechanisms all important to ensure the smooth functioning of these systems when either a server crashes or due to some other type of failures. In-memory data management systems commonly employ both two fundamental methods for persistence and recovery: snapshots and log-based methods, though the log-based systems are slightly more frequent. For instance, RAMCloud uses a logging approach similar to log-structured file systems, and Redis provides both snapshot and logging options where snapshots is the default setup [5]. In a few research articles, emphasis has been put on optimizing the scalability of logging mechanisms [6]. Specifically, Memcached is designed as an in-memory-only caching system with no persistence, and MongoDB uses an approximate in-memory design and memory-mapped files. In particular, most researching into data persistence and recovery have focused mostly on SQL transactional frameworks.

However, with the increased adoption of IM-NoSQL in real applications, it often becomes a necessity for the developers and users to make the right choice in terms of persistence and recovery configurations so that their NoSQL system is reliable. This work presents a full performance evaluation of persistence and recovery mechanisms within IM-NoSQL frameworks over a variety of workloads. We analyse the efficiency of a number of major persistence and recovery methodologies in NoSQL-server failure scenarios, focusing particularly on snapshotting and logging methodologies. Our investigation covers all the fundamental components of NoSQL workloads involving execution, persistence, and recovery operations. In this research, however, we focus on Redis because it has emerged to be among the best implementations of IM-NoSQL technology. Most studies on persistence as well as recovery in databases have addressed traditional SQL-based transactional systems much more purely. This work is actually the first rigorous comparative study of different persistence models, their influence on the performance of conventional NoSQL workloads, and their impact on recovery efficiency in NoSQL-oriented in-memory systems. Moreover, given the advent of byte-addressable, non-volatile memory (NVM), which revolutionized the design of memory systems, experts believe that it is a very appropriate time for reassessment of persistence and recovery strategies.

1.1 Motivation

In-memory databases (IMDBs) have emerged as a pivotal technology within the data management paradigm, propelled by the escalating demand for rapid data access and real-time processing capabilities. Conventional disk-based databases frequently encounter challenges in fulfilling the performance expectations of contemporary applications, particularly within sectors such as finance, e-commerce, and real-time analytics. IMDBs address the above challenge mainly by preserving memory data, which in return much boosts the read and write processes speed compared to conventional storage solutions [9].

The architectural framework of in-memory databases generally utilizes a key-value store model, which enhances the speed of data retrieval and manipulation processes. There are examples of in-memory databases like Redis, Memcached, and Apache Ignite, each focusing on a specific aspect that finds specific use applications.

Though having better performance considerations, the presence of persistence and recovery systems in in-memory databases poses big challenges. Contrary to the conventional databases, which natively maintain data on disk, IMDBs have to take potent methods ensuring the persistence of data in systems crash, crashes, or unexpected termination [8]. The need for persistence has, therefore motivated the development of various techniques, including:

- **Append-Only File (AOF) Logging:** AOF logging, records every write operation to a file, enabling the database to reconstruct the in-memory state during recovery [1]. Although this methodology offers robust durability assurances, it may introduce performance overhead during write operations.
- **RDB (Redis Database) Snapshotting:** Periodically, RDB Snapshots will archive the current state of the database to disk, which provides quicker recovery. However risk due to potential data loss between snapshots jeopardizes this strategy's balance between performance and durability, so it needs careful management of snapshot interval [6].
- **Hybrid Approaches:** Some modern in-memory databases take the best of both approaches by combining AOF and RDB methods to optimize performance as well as safety for the data. Such systems are able to enhance recovery performance without increasing latency in regular operations by maximizing the efficiency of multiple persistence models [10].

Recent research has been directed toward enhancing these persistence mechanisms to achieve higher throughput, lower latency, and optimized recovery procedures.

Research work shows that sophisticated techniques such as batch writing significantly reduce the I/O load associated with conventional persistence approaches and contribute to higher system throughput [2]. Progress in memory management techniques and caching methodologies has made the system much effective in handling large-scale datasets of In-memory databases. It examines the present state of in-memory databases, explaining different persistence and recovery strategies designed. By synthesizing extant literature and identifying salient trends and deficiencies, this paper aspires to comprehensively understand how these systems can be optimized for high-performance computing environments.

1.2 Structure of the Paper

The framework of this survey research manuscript is designed to deliver an extensive examination of developments made in in-memory databases, with particular emphasis on the persistence and recovery mechanisms that improve their efficiency. Building on foundational work, such as Persistence and Recovery for In-Memory NoSQL Services: A Measurement Study (Xianqiang Bao, June 2016) [15], which evaluates the effectiveness of different persistence models in Redis [7], this paper is organized into several key sections. The introduction section mentions the shortcomings of traditional databases and the need for high-performance substitutes in such real-time applications. The literature survey amalgamates all the recent research currently available on In-memory NoSQL systems and their varied methodologies for persistence; thus, it provides strong context for the survey. We then identify emerging trends and research gaps in the topic, highlighting areas that need exploration. The subsequent sections discuss the pros and cons of the persistence mechanisms in use - AOF logging and RDB Snapshotting [1] [6], as well as innovative techniques like batch writing that aim at optimizing I/O operations. The paper concludes with a summary of key findings, Implications for practice and suggestions for future research. This kind of outline demonstrates how the survey synthesizes existing knowledge and offers insights that can guide further advancements in in-memory database technologies.

2 Literature Review

We describe here five types of persistence mechanisms and two types of recovery mechanisms that are most popular in IM-NoSQL.

1.1 The Persistence Models

Current IM-NoSQL data persistence requirements are usually dependent on the hosted applications' data persistence requirements. For some systems, there is no persistence at all. For example, Memcached is a widely used distributed in-memory cache system. Here, memory acts as a data cache. Therefore, its design and implementation had not included any form of persistence support [5].

Several other in-memory systems, such as Redis, provide good implementations of persistence in a variety of user-friendly formats. From all the available approaches, Snapshots and logging are the most common where each change to the in memory data is persisted to a storage system [1].

Snapshot Persistence: This is a snap-shot model in an IM-NoSQL system taking a picture of all the working datasets stored in memory at regular time intervals. Then this picture is saved into the storage as the latest snap shot file [2]. The time intervals are usually determined by a number of seconds, like 900 or 300 seconds. In addition, other NoSQL systems support a snapshot triggering condition based on other parameters, such as the number of writes that have occurred (for example, every 10 writes, or every 100 writes). The process of taking a snapshot is split into two stages that could potentially lead to creating persistent snapshots: Step 1. Snapshot trigger: This step determines when to take a snapshot. This means that when the triggering condition is frequent in a given number of writes, the trigger manager operates in continuous or periodic cycles. During these cycles, it queries the update profiling manager, which observes the number of write requests made and calculates the time interval to assign to each write based on either the last taken snapshot or when the system has been operational. Once the process determines there have been sufficient writes within a specified time period, the process will commence with booting up the snapshot and commencing saving of it (flush). The read/write APIs are usually represented as get/set for NoSQL. So, the counts of write requests used by the snapshot trigger can directly be retrieved from the count of invoked sets. (2) Snapshot Save: When the snapshot action is triggered, the NoSQL system saves the snapshot to the storage and keeps it as the latest snapshot file. In more detail, the system creates both child and parent processes. The child process will start taking snapshots and begins saving the working dataset in the permanent storage to a temporary snapshot

file. The parent process will continue with the normal activities of the NoSQL system. When the child process is done writing the last data record of the temporary snapshot file, it will replace the old version with a new one. The child process will thus use the copy-onwrite method and copy the in memory structured dataset on disk to preserve the temporary snapshot file so that the parent process can accept the write requests without waiting for it to finish. The primary example only needs to be throttled during a replacement of the old snapshot file. The in-memory structured dataset is usually compressed before it is actually saved in order for I/O work better for storage and also for saving space.

Log Persistence: If persistence is achieved using the Log model, for each write request that it receives in the server, the system updates it as an entry of the log record, which is then stored in the log buffer before conducting the update operation. Upon crossing a certain threshold, the logging process is triggered, and the log buffer flushes to a log file located in the persistent storage [3]. The logging trigger threshold determines exactly the right time to flush the log buffer; hence, the configuration of the threshold is critical to the performance of any writes-intensive workload and, importantly, to the preservation of the dataset currently being updated. In other words, increasing the threshold lowers the rate of log flushes to persistent storage, which in turn minimizes the performance penalty imposed by Logging on the basic NoSQL operations, while simultaneously weakening the guarantee that the currently updated dataset would be persisted. The log model is extensible and can be extended in a number of ways or even be made persist at arbitrary levels of granularity. Below we describe three modes of implementation. (1) Direct flush log, Log-Immediate: With each log record creation after the NoSQL server has received the write request, the logging process flushes the log record to the log file. This Log-Immediate persistence model triggers the logging on every write request. This makes each write request even more latency-expensive in Log Immediate because, based on the WAL principle, the server has to wait for its corresponding log record to be appended to the log file before considering the request as accomplished. As mentioned earlier, this degrades the performance of the system the worst when the log files are stored on a slow persistent storage media like HDD, since disk I/O can easily become a bottleneck. One benefit of flushing right away is it requires the logging procedure to reserve a minimum log buffer. (2) Periodical flushing (Log-Periodical): The logging mechanism periodically checks the log buffer at regular preset log append interval configured in the system configuration such as one second in Redis. Since the log records are being appended to the log buffer, the logging mechanism will process contents in the log buffer and transfer them to the log file via procedures called log-append. Therefore, it monitors both the temporal interval and the condition of the log buffer. Under Log-Periodical, it treated the write request to have completed only when its corresponding log record has been successfully appended to the log buffer. It

also used the mechanism of copy-on-write in order to dump the log records into a log file. Under this model, it satisfies the need for the guarantee of WAL. (3) Log-Deferring, Deferring Flush This step typically consists of two separate steps: in the first step, logging, the mechanism of logging is producing log records consistently to help maintain a listing of all incoming updates in the log buffer that resides in memory and then leaves it up to the operating system to handle flushing the log buffer. Thus, it is assumed that the operating system monitors the flushing of both of its buffers-writing buffer and log buffer-to persistent storage. Just like in Log-Periodical, at the instant at which a write request, under the Log-Deferred model also attains log record's successful addition to a log buffer, the request is also treated as completed. The above framework has the same weakness as does the one of Log-Periodical. If the OS does not flush its log buffer before the impending server crash, then the persistence guarantee lasts only until the last such flushing. Log-Immediate takes the minimum flushing interval for the log buffer, but gets the highest working dataset persistence. Still, it is the most expensive because it considerably reduces write performance, and this is because it needs to flush immediately with each one of its write requests targeted at the slow I/O storage subsystem. Because Log Periodical and Log-Deferred both run log records in batches, the length of Log-Deferred is dependent upon the operating systems' ability to cope with the draining of the write buffer, potentially causing it to be a length many times greater than that of the Log-Periodical interval. The flushing of the log buffer should be the longest time and thus falls on the persistence level minimum, but Log-Deferred shows very high write throughput. Conversely, Log-Period is a blended approach aimed at striking equilibrium between high persistence and significant write performance, especially in comparison to Log-Immediate, which favors high persistence behind performance, as well as Log-Deferred, which favors better performance behind lower levels of persistence. Therefore, to obtain high persisting with acceptable data loss risk, IM-NoSQL will likely turn into Log Periodical.

No Save without Persistence: To gain a better understanding of the various snapshot and logging-based persistence schemes, we also include the No Save model as a very simple baseline that provides no form of persistence support whatsoever. In the No Save configuration, all datasets involved in the operational data set-that is, database information and index data -are retained entirely in memory. It can use its allocated swap memory if the available dataset exceeds the physical RAM but falls into the swap space of an operating system . A major limitation of the No Save model is that if the system crashes, which is similar to analogous things, all datasets operational in the physical RAM and the swap space will be lost without any possibility of retrieval in case of system shutdown because this No Save model doesn't support persistence. No Save shows higher throughput compared to other snapshot or logging persistence models, especially in a scenario where the runtime environment runs under serious

competition for resources with resultant performance bottlenecks. The case in point is when the working dataset goes beyond the amount of physical RAM available, with complications such as OS swapping or page faults.

2.2 The Recovery Models

Snapshot-based Recovery: In such scenarios where the targeted NoSQL database necessitates a server instance restart following a failure or shut down, the snapshot recovery model demands that the server implement two-phase recovery of the dataset from the latest snapshot file:

- (i) It first loads the full snapshot file to memory, and
- (ii) It then rebuilds the working datasets [8]

Loading the snapshot file: When there is a server instance restart, the server establishes the configuration of the snapshot persistence model, and the latter triggers the calling of the recovery process based on the snapshot. Determination of the snapshot file and evaluation of whether or not it is complete, if it corresponds to the correct file or version of the current instance, and other such factors defines the first stage of recovery. Then, the loading of the snapshot file using streamed reads is initiated by the recovery process; usually, the snapshot file is in the binary format and loads every time either one key-value formatted record or a batch of such records into the memory. (2) Recovery of the operational dataset: Once all records have been recovered into memory, any compressed records will be decompressed first. Then the recovery procedure continues injecting these records into the newly created in memory data structure, e.g. hash table. During this process, it also builds an index for these records. In situations where the snapshot file is large but remains within the size of manageable memory with the swapping capabilities of the operating system, the operating system's memory swap area becomes relevant during page faults. The frequent swapping of pages in memory may degrade the efficiency of the reconstruction process. However, when the size of snapshot file is much bigger than memory capacity for processing, NoSQL systems will prompt an out-of-memory error, since recovery is always called by using available memory resources, which also contain the swap space of operating systems. Therefore, the operation reconstituting the functional dataset will be shut down in the recovery process. So, it is necessary to check memory capacity against the size of the snapshot file. The NoSQL server instance may only restart processing incoming workload requests once the whole reconstruction of an entire dataset has been completed.

Log-based Recovery: After starting the restart process for a NoSQL server by an approach of log-based persistence, such as Immediate, Periodical or Deferred, a two-step procedure enables the log-based recovery mechanism to rebuild the complete

operational dataset by replaying the entire entries in the log: (1) load log file-a function akin to loading a snapshot file; and (2) replay log records, in which the NoSQL server depends on the recovery client, typically implemented within the server's configuration for log-based persistence, in processing the write commands as well as reconstructing the key-value records [6]. This includes extracting the operands from the log record entry regarding the write command and executing the write command so as to rebuild the corresponding key-value record. After the recovery client processes all the log entries, the entire complete dataset is reassembled. Further, if the log file is large and exceeds the memory of the NoSQL server, the recovery process could drop dramatically in performance and may not even fully recover completely.

3 Comparative Analysis

In this section, we conduct a comprehensive comparative analysis to evaluate the performance implications of different persistence configurations in Redis, specifically focusing on Append-Only File (AOF) and Snapshot mechanisms. These comparisons aim to assess how various persistence configurations impact Redis throughput and latency under different client loads. By understanding the trade-offs between performance and data durability, we can provide insights on configuring in-memory NoSQL databases for optimal performance in environments where resilience, speed, and stability are critical.

To ensure this analysis, we generated a series of graphs to visualize the effects of three primary configurations:

1. **Merged AOF and Snapshot:** In this configuration, we individually compare the throughput and latency effects of enabling AOF or Snapshot.
2. **No Persistence:** This configuration, where both AOF and Snapshot are disabled, shows Redis's performance without persistence overhead.
3. **Both AOF and Snapshot Enabled:** This configuration shows the combined impact of enabling both persistence mechanisms simultaneously. Each graph set includes two primary metrics:
 - **Throughput (requests per second):** Reflects Redis's ability to handle client load efficiently.
 - **95th Percentile Latency (in milliseconds):** Indication of response times under different configurations, mainly focusing on higher-latency outliers that could impact user experience under peak load conditions.

- No Persistence:** Represented by a single red line, this baseline graph shows Redis's maximum performance when persistence is entirely disabled. In the Throughput graph, Redis achieves the highest throughput here, as it is not impacted by the persistence overhead. Similarly, the Latency graph shown in Fig. 1. minimal latency in this configuration, reflecting the raw speed achievable without persistence.

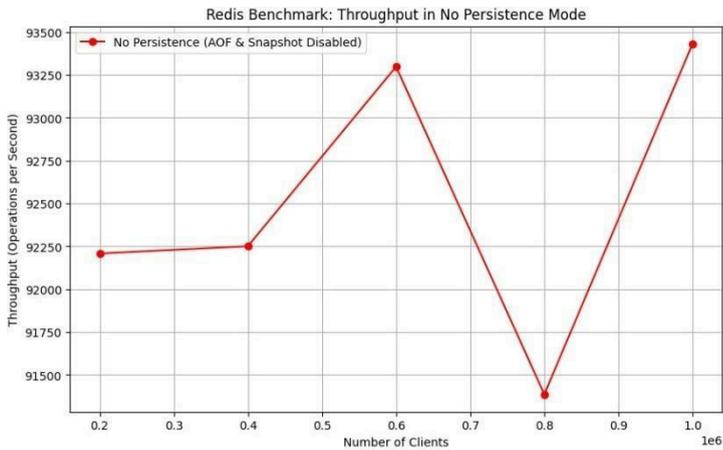


Fig 1 Redis Benchmark for No-Persistence Throughput.

- Merged AOF and Snapshot:** This graph set shows two lines, with blue representing AOF and green representing Snapshot. It compares the throughput and latency of each configuration when only one persistence method is enabled. The Throughput graph illustrates how enabling AOF leads to slightly higher throughput than Snapshot, though this comes at a higher latency cost. The Latency graph shown in Fig. 2. that Snapshot maintains relatively lower latency, suggesting that it offers a more balanced trade-off for scenarios requiring moderate durability without a significant performance penalty.

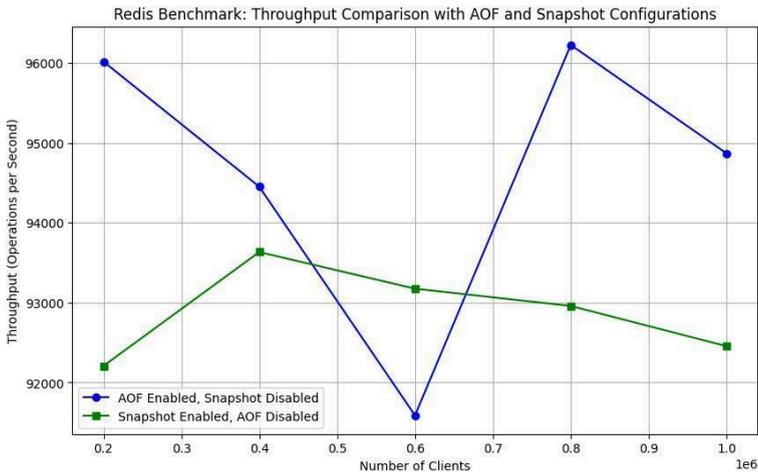


Fig 2 Redis Benchmark for AOF Vs Snapshot Throughput

- 3. **Both AOF and Snapshot Enabled:** This graph set uses dashed lines to show the performance impact when both AOF (blue) and Snapshot (green) are enabled together. The Throughput graph shows that throughput decreases moderately when both persistence mechanisms are active, as the overhead of dual persistence is added. In the Latency graph (Fig. 3.), we observe that enabling both mechanisms leads to higher latency, especially for AOF, which consistently has higher 95th percentile latency than Snapshot. This demonstrates the compounded impact on performance when durability requirements demand dual persistence.

4. Combined Graph Analysis: The combined graph brings together all three configurations—Merged (AOF and Snapshot), No Persistence, and Both AOF & Snapshot Enabled—into a single view, allowing for a side-by-side comparison. In the Throughput graph, the red line (No Persistence) consistently outperforms all other configurations, while the blue and green dashed lines (Both Enabled) show the lowest throughput due to the combined persistence overhead. The solid blue and green lines (Merged AOF and Snapshot) fall in between, indicating the trade-off between having one persistence mechanism versus none or both.

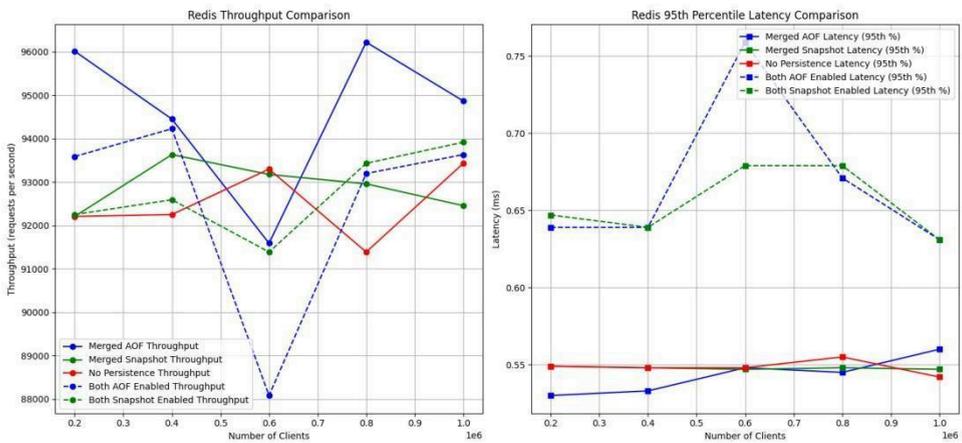


Fig 3. Redis Benchmark for AOF & Snapshot Enabled Throughput and Latency.

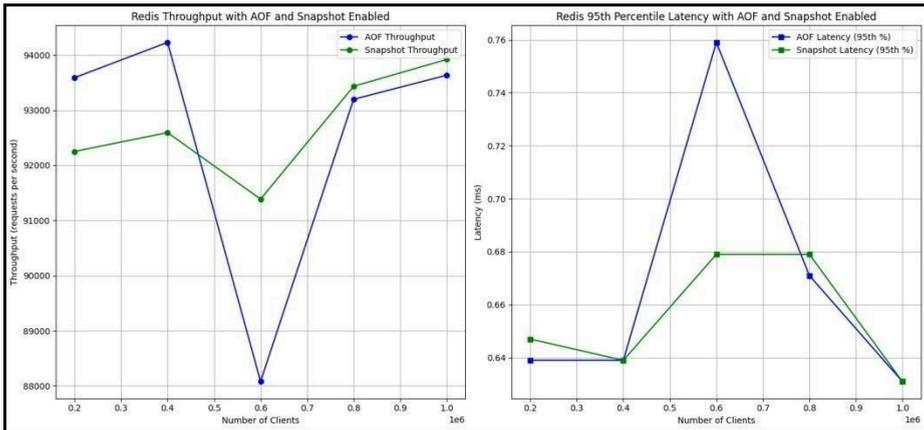


Fig. 4. Redis Benchmark for Combined Analysis of Models Throughput and Latency.

In the Latency graph shown in Fig. 4. the No Persistence line show the lowest latency, with Merged configurations in the middle, and the Both Enabled configuration with the highest latency values, particularly for AOF. This combined graph clearly illustrates how performance and durability vary across configurations, providing valuable insights into how Redis can be tuned to prioritize either throughput, latency, or data durability.

4 Conclusion

Our research on In-memory NoSQL databases (IM-NoSQL), specifically Redis, reveals several limitations and performance drawbacks associated with using persistence mechanisms like Append-Only File (AOF) logging and snapshots. Firstly, enabling persistence options such as AOF and snapshots, while beneficial for data durability, has a noticeable impact on performance. Both mechanisms increase memory usage and can reduce throughput, especially under high client loads. The performance cost of maintaining persistence means that even with optimized configurations, there is a measurable latency overhead compared to running Redis without persistence.

Another significant drawback is the heavy reliance on DRAM capacity when persistence is enabled. Snapshots require considerable amount of memory, which can lead to memory swapping when the dataset size approaches the DRAM limit. This slows down performance and risks potential data access delays, as the system may have to manage memory more aggressively to maintain functionality.

Additionally, recovery from a crash can be inefficient when AOF logging or snapshots are enabled, particularly with large datasets or extensive AOF logs. The time required to restore the data state depends heavily on the available DRAM, and any memory shortage will further slow the recovery process. This dependency on high memory resources makes IM-NoSQL databases less resilient in resource-constrained environments.

In conclusion, while AOF and snapshots provide essential durability features, their performance costs, memory demands, and slow recovery times can hinder the overall efficiency of IM-NoSQL databases like Redis. These findings suggest that in scenarios where high throughput and minimal latency are critical, persistence mechanisms should be carefully considered, as they may compromise the expected performance and efficiency of the database system.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [1], an LNCS chapter [2], a book [3], proceedings without editors [4], as well as a URL [5].

References

1. X. Bao, L. Liu, N. Xiao, Y. Lu, and W. Cao, "Persistence and Recovery for In-Memory NoSQL Services: A Measurement Study," 2016 IEEE International Conference on Web Services (ICWS), San Francisco, CA, USA, 2016, pp. 530-537, doi: 10.1109/ICWS.2016.74.
2. J. Huang, K. Schwan, M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. Proc. of the VLDB Endowment, Vol. 8, No. 4, 2014.
3. J. Arulraj, A. Pavlo, S. R. Dullloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. ACM SIGMOD 2015, pp.707–722.
4. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al. Bigtable: A Distributed Storage System for Structured Data. USENIX OSDI 2006, pp.205–218.
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6), 205-220..
6. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (pp. 143-154).
7. Bao, X., Liu, L., Xiao, N., Zhou, Y., & Zhang, Q. (2015, June). Policy-driven configuration management for NoSQL. In *2015 IEEE 8th International Conference on Cloud Computing* (pp. 245-252). IEEE.
8. Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., & Rosenblum, M. (2011, October). Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 29-41).
9. T. Wang and R. Johnson. Scalable Logging through Emerging Non-Volatile Memory. VLDB 2014, vol.7.

10. Zhang, H., Chen, G., Ooi, B. C., Tan, K. L., & Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7), 1920-1948.
11. Dragojević, A., Narayanan, D., Castro, M., & Hodson, O. (2014). {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (pp. 401-414).
12. Lim, H., Han, D., Andersen, D. G., & Kaminsky, M. (2014). {MICA}: A holistic approach to fast {In-Memory} {Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (pp. 429-444).

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (<http://creativecommons.org/licenses/by-nc/4.0/>), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

